Report #4
PD-Internship
Lerner Paul
04/04/2018

These last 2 weeks I've read about RNNs and Time series analysis as planned. I also started making some experiments based on my reads. This report is divided into 2 sections : 1 Literature Review and 2 Code. As usual I highlighted the parts where something is unclear or I doubt and **need feedback**…

Summary

Lexicon **1**
1 Literature Review **2**
1.1 Model Architectures et al. 2
1.2 Hyperparameters finetuning 6
1.3 Regularization & Overfitting 8
1.4 Transfer Learning 10
1.5 Conclusion 11
2 Code **12**
2.1 Data preprocessing 12
2.2 Model architecture & evaluation 14
2.3 Visualization & Interpretation 17
2.3.1 Input weights 17
2.3.2 Forget Gate 18
Conclusion - Todo List **18**
References **19**

Lexicon

Positive Predictive Value (PPV, aka precision) is the proportion of patients with positive test results who are correctly diagnosed.

Negative Predictive Value (NPV) is the proportion of patients with negative test results who are correctly diagnosed. (Altman & Bland).

$$PPV = \frac{TP}{TP+FP} \quad ; \quad NPV = \frac{TN}{TN+FN}$$

Mean *f1-score* (allows for comparison between classification on different number of class and imbalanced datasets) :

$$F_m = \frac{2}{|c|}\sum_c \frac{\text{prec}_c \times \text{recall}_c}{\text{prec}_c + \text{recall}_c}$$

Where c is the number of class, $prec$ is the PPV and $recall$ is the Se.

Fully Connected (FC) Layer : layer of neurons which are fully connected to the previous layers' neurons. Aka :

- linear layer, for the PyTorch users
- dense layer, for the TensorFlow users
- feedforward layer, as opposed to a recurrent layer

1 Literature Review

Disambiguation : All the works cited below use a RNN or a LSTM.

Salehinejad et al. wrote a thorough review about RNN, they present :

- many different ways of training a RNN other than stochastic gradient descent.
- many different architectures among which, Hierarchical Subsampling RNN caught my attention (cf. below).
- 4 domains in which RNN have made a great impact, namely : text, audio & speech, image, and video processing.

1.1 Model Architectures et al.

3 ways of making a Deep RNN (Pascanu et al. 2013) :

- Deep Input-to-Hidden Function, e.g. word embeddings
- Deep Hidden-to-Hidden Transition
- Deep Hidden-to-Output

The authors argue that stacked RNN, which allow the model to deal with multiple time scales in the input sequence, are shallow as the transition between the different hidden layers is

linear. The authors ran an experiment and found that deep transition and deep output & transition RNNs outperformed both the conventional RNN and the stacked RNN on the task of language modeling, although all four models achieved similar results on Polyphonic Music Prediction. However the authors mentioned the difficulty of training deep output & transition and stacked RNN. I wasn't able to find any other authors who use Deep Hidden-to-Output, moreover, the authors didn't experiment with the deep output model (only with the deep output & transition). They don't explain why.
Sønderby et al. make use of the Deep Hidden-to-Hidden Transition to predict protein secondary structure from the amino acid sequence. They don't really motivate this choice or explain their better results (they outperform SoA) except by citing Pascanu et al.
Deep Input-to-Hidden Function is not suited for our data, as the variables are "continuous" (as opposed to discrete like words).
Pascanu et al. 2013 don't talk about adding layers after the last hidden layer of a RNN, like Dai et al. (see below).

Greff et al. 2015 didn't find any statistical difference in the performance of LSTM, peephole variant and GRU on tasks speech recognition, handwriting recognition, and polyphonic music modeling.

We should use a bidirectional LSTM (bLSTM) which allows for use of future and past context to interpret the input at timestep t. Forward LSTM are more relevant for real-time applications, as we don't know the future yet (Hammerla et al. 2016). Hammerla et al. 2016 compared several deep learning models (including LSTM) in a task of Human Activity Recognition based on several sensor data such as accelerometer data. They make use of 3 different dataset for comparison. It's interesting to note that their data was either sampled at 30 Hz or downsampled around 30 Hz (*"in order to have a temporal resolution comparable"* to the 30 Hz dataset). This is way less than **PaHaW** which is sampled at 200 Hz. In the same way, Greff et al. 2015 used the IAM-OnDB (very similar to ours) and splitted the data into lines (i.e. one sequence corresponds to one line) and subsampled this sequence to half of its length, which *"speeds up the training and does not harm performance"*.
The authors trained each model at least 30 epochs and for a maximum of 300 epochs (depending on the validation performance).

Graves, 2012 introduces Hierarchical Subsampling RNN which breaks the sequences into subsamples. The goal of this model is to handle long sequences. The model is a stacked LSTM where each layer gets smaller and smaller : each layer is trained over each subsample and sums the hidden states of the subsamples before feeding it to the next layer (cf. figure below). The model won handwriting recognition competitions in three different languages at the 2009 International Conference on Document Analysis and Recognition. Although I like the theoretical aspect of this paper, it may be relatively hard to implement and I wasn't able to find reviews comparing this model to other RNNs (e.g. Greff et al. 2015 don't try it).

Moreover, the model achieved pretty poor results on the Phoneme recognition results on TIMIT (28% error rate). The same author later achieved 18% error on the same dataset with a Stacked bLSTM (Graves et al. 2013).

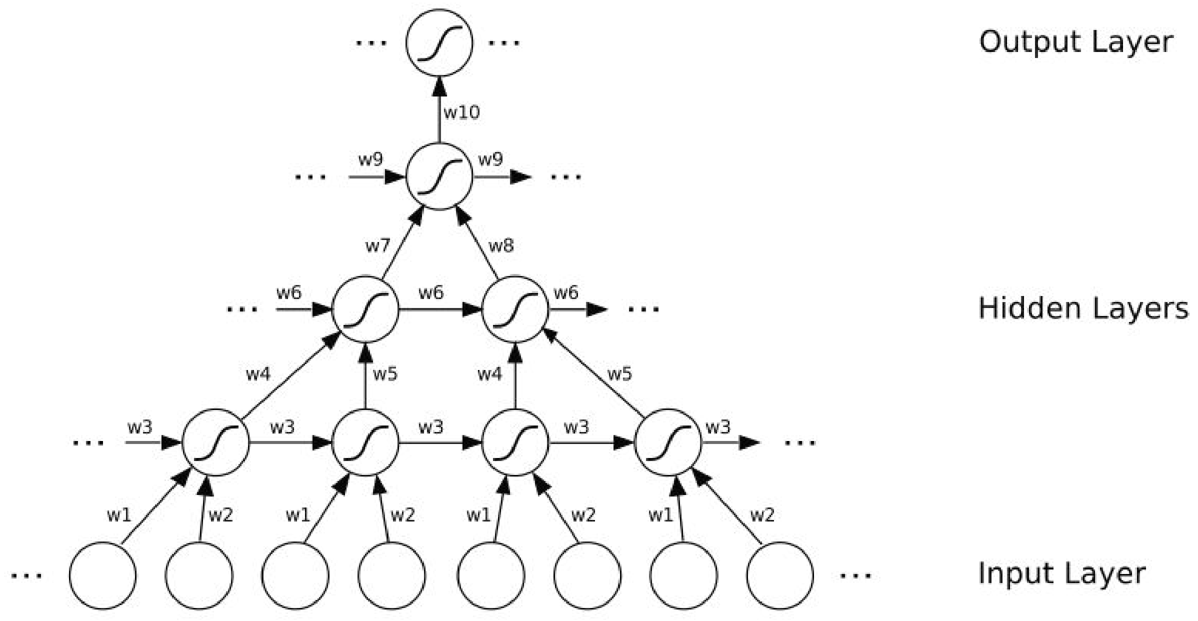
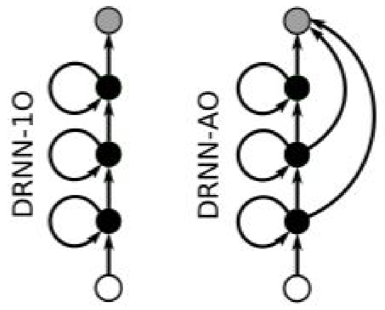


Fig. 9.2 An unfolded HSRNN. The same weights are reused for each of the subsampling and recurrent connections along the sequence, giving 10 distinct weight groups (labelled 'w1' to 'w10'). In this case the hierarchy has three hidden level and three subsampling windows, all of size two. The output sequence is one eighth the length of the input sequence.

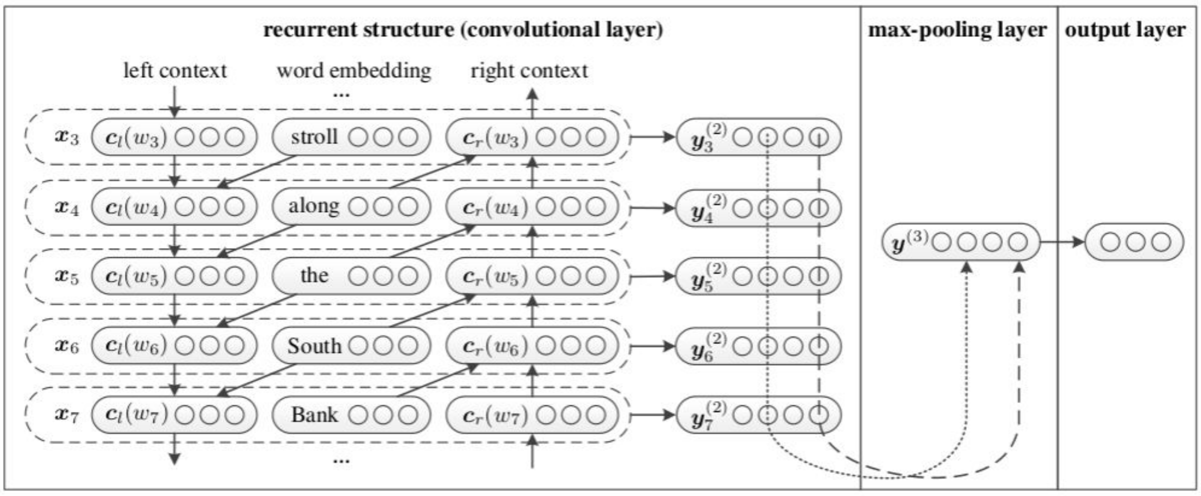
Yao et al. introduce the Depth-gated RNN which connects cell states between the different layers of a stacked LSTM. The authors outperform state of the art on Machine translation and Language modeling. However the paper is very short and doesn't explain the better results of their model.

Hermans & Shrauwen experiments with a stacked RNN (which they call a deep RNN, unlike Pascanu et al.) on a character-level language modeling task and show that one can train a very large stacked RNN (5 layers of 727 units → approximately 4.9 million trainable parameters) using only stochastic gradient descent and the gradient clipping trick. They also show that each layer of the stacked RNN works on different timescales. They tried 2 types of stacked RNN : DRNN-1O where only the last layer is fed to the output and DRNN-AO where all the layers are fed to the output (cf figure below).



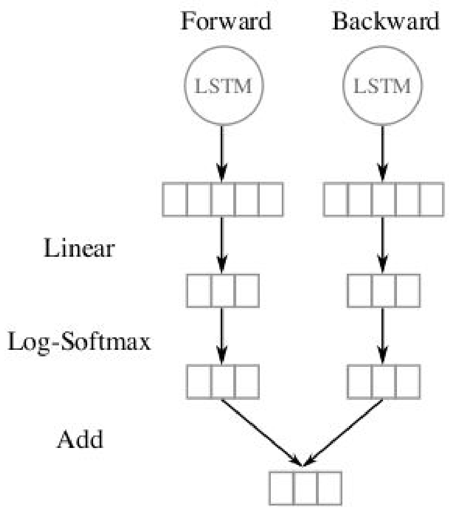
They had slightly better results with DRNN-1O and suggest that the 2 networks don't work on the same timescales depending on their layers : *"The DRNN-AO has very short time-scales in the three bottom layers and longer memory only appears for the two top ones, whereas in the DRNN-1O, the bottom two layers have relatively short time scales, but the top three layers have virtually the same, very long time scale."*

Lai et al. use a quite strange model for text classification (topic classification, sentiment classification and writing style classification). They use a bidirectional RNN (not LSTM) to capture the context of a word then concatenate the both directions outputs and the word embedding in a vector which they feed to a fully connected (FC) layer. They then max-pool over this layer to have a vector that represents the whole sequence, which I find very strange since the last output of a RNN should already contain information about the whole sequence. They add a last FC after the max-pooling for classification, the figure below summarizes their model :


recurrent structure (convolutional layer)
max-pooling layer
output layer
left context
word embedding
right context
x_3 $c_l(w_3)$ stroll $c_r(w_3)$ $y_3^{(2)}$
x_4 $c_l(w_4)$ along $c_r(w_4)$ $y_4^{(2)}$
x_5 $c_l(w_5)$ the $c_r(w_5)$ $y_5^{(2)}$
x_6 $c_l(w_6)$ South $c_r(w_6)$ $y_6^{(2)}$
x_7 $c_l(w_7)$ Bank $c_r(w_7)$ $y_7^{(2)}$
$y^{(3)}$

The authors outperform or reach state of the art on 3 out of 4 datasets.

Chiu et al. 2016 use a bLSTM but instead of summing the hidden states of the two LSTMs they connect each of it to a fully connected layer then to a softmax layer. The two softmax layers are then summed as in the figure below (Chiu et al. 2016) :



The authors made this choice after *"preliminary experiments"*. Their classification task is Named Entity Recognition.

Ordóñez & Roggen 2016 introduce a Deep Convolutional LSTM, which would fall inside the "Deep Input-to-Hidden Function" category, I guess. Their task is Human Activity Recognition. They use a sliding window to perform convolutions which I find strange since it "breaks" the sequence. However, they feed the sliding windows through convolution layers that act as feature extractor then feed it to the LSTM. They use a stacked LSTM with 2 layers (because of Kaparthy et al.). They compare this model to a regular CNN with the same architecture, except that the LSTM layers are replaced by FC layers. The authors outperform state of the art on 2 different datasets and obtain better results with the Convolutional LSTM than with the regular CNN.

Dai et al. 2015 (cf. 1.4 Transfer Learning) truncate the backprop of their LSTM to 400 time steps to allow the model to learn (i.e. fit the data). The authors also use a hidden layer of 30 units with dropout of 50% between the last hidden state and the classifier.
Like them, Malhotra et al. (2016, June) used a LSTM Seq2Seq but they use it to detect anomaly in time-series. The model is first trained to reconstruct normal time-series and then uses reconstruction error to detect anomalies. The authors don't compare their model to the state of the art so it's hard to have an opinion of it, it's seems a little far-fetched to me, a priori.

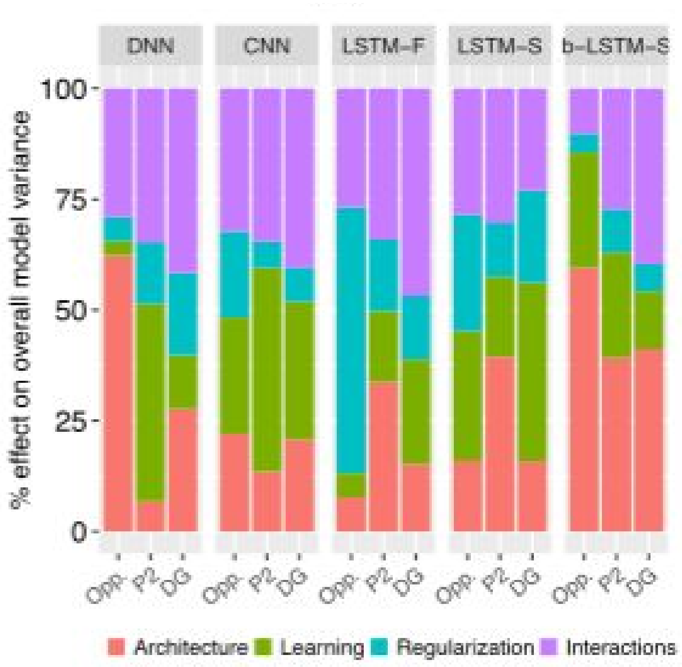
1.2 Hyperparameters finetuning

Although trained on very different classification task than ours, Jozefowicz et al. 2015 found that initializing the bias of the forget gate to a large value such as 1 or 2 significantly improved performance over all tasks "*(a practice that has been advocated by Gers et al.*

(2000))". In PyTorch, the forget gate bias is stored as ***b_hf*** in ***bias_hh_l[k]*** : the learnable hidden-hidden bias of the *kth layer (b_hi|**b_hf**|b_hg|b_ho)*, of shape *(4*hidden_size).*

Greff et al. 2015 wrote a nice introduction to LSTM and its use in the different fields of Machine Learning. They show that optimal value for the learning rate is dependent on the dataset and that, independently of the dataset, there is a large basin (up to two orders of magnitude) of good learning rates inside of which the performance does not vary much. Thus, the authors advise to search for learning rate on a logarithmic scale and to stop when the performance has decreased. Moreover, they show that momentum doesn't affect training time nor performance, therefore we shouldn't spend time fine-tuning this parameter.
The authors conclude that *"hyperparameters can be treated as approximately independent and thus **optimized separately**."*

Figure from Hammerla et al. 2016 explaining the variance on a model performance (mean f1-score) on different dataset (represented by the columns here) depending on it's hyperparameters :



Interactions corresponds to interactions between the different hyperparameters categories. Please focus on the right-most group of columns which corresponds to the bidirectional LSTM which we plan to use. We can see that Architecture hyperparameters explain almost 50% of the variance in all 3 datasets. Notice that this is only true for this model and it is pretty crazy as the only Architecture hyperparameters that the authors changed in this model is **the number of units** of the layers, unlike for the other models where they also changed the number of layers (and number of kernels and kernel width for the CNN). Therefore, we should first focus on tuning this parameter before the others (the authors recommend to do so). These results goes against those of Greff et al. 2015 who show that learning rate accounts for 67% to 91% of the model performance depending on the dataset. This could be explained because Hammerla et al. only searched between 10^{-3} and 10^{-1} whereas Greff et al. searched between 10^{-6} and 10^{-2}. However, I can't find explanation for the fact that, in the work of Greff et al., the number of units of the layers only explains between 1% and 10% of the model performance, depending on the dataset. Therefore, I think maybe the work of Hammerla et al. may be too dependent of their classification task.

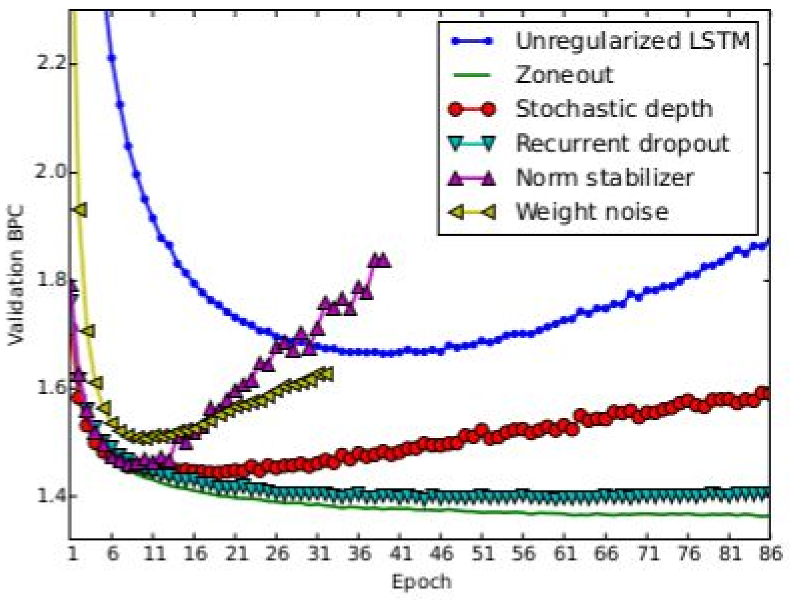
1.3 Regularization & Overfitting

All the literature seemed to agree that dropout should not be added between recurrent connections as it would disable the sequence modeling of the network (Pham et al.) until 2015 (Krueger et al.). Pham et al. worked on handwriting recognition and show that *"The word recognition networks with dropout at the topmost layer significantly reduces the [Character error Rate] and [Word error Rate] by 10-20%, and the performance can be further improved by 30-40% if dropout is applied at **multiple LSTM layers**."*. So we should not only add dropout after the LSTM but also between the different LSTM layers (if we use a stacked LSTM).

Cooijmans et al. argue that recurrent batch normalization (RBN) is suited to regularize RNNs. Batch normalization consists in standardizing the weights of each layer to have a mean and variance similar to a Gaussian variable. Like dropout (see below), it was thought that batch normalization should only be applied to non-recurrent layers. The authors argue that this was because of a bad initialization of the γ parameter, which is used in the batch normalizing transform. They outperform state-of-the-art on the pixel by pixel MNIST classification tasks and CNN question-answering task, fall behind on Penn Treebank test sequence and text8 test sequence but RBN always improves performance over the baseline LSTM. See Krueger et al. below for more experiments.

Semeniuta et al. introduced *recurrent dropout*, which applies dropout only to the cell update vector g_t. The authors argue that this approach *"considers the architecture as a whole with the hidden state as its key part and regularize the whole network."*. The authors experiment on three widely adopted NLP tasks: word- and character-level Language Modeling and Named Entity Recognition. They show that *recurrent dropout* can be coupled with forward dropout (cf. Krueger et al. below for more experiments).

Krueger et al. introduce *zoneout*, which forces some values of the hidden state and cell state to *maintain* their previous values (as opposed to *dropout* where the values are set to zero). They apply it on recurrent connections.
Figure comparing different regularization techniques on character-level language modeling on Penn Treebank :



The BPC measures the entropy of the model : it's the Negative log-likelihood divided by the natural logarithm of 2.
Norm stabilizer is another technique introduced by the *Krueger* & Memisevic in 2015 (same 1st author).
Consistently with Greff et al., we can see that weight noise is not a very good regularizer.
Stochastic depth drops entire layers of feed-forward residual networks which is equivalent here to zoning out all of the units of a layer at the same time.
Recurrent dropout (Semeniuta et al.) and zoneout are the best regularizers on this dataset. Moreover, on the world-level task of the same dataset, the authors don't achieve competitive results without recurrent dropout (i.e. only with zoneout).
Strangely, the authors experimented with recurrent batch normalization (RBN) on pMNIST's digit classification task but not on the above experiment. They found that RBN outperformed zoneout and that merging the 2 techniques achieved slightly better results than RBN alone.
The authors found that zoneout helped with vanishing gradients through an experiment on pMNIST's digit classification task : *"zoneout propagates gradient information to early timesteps much more effectively than dropout on the recurrent connections, and even more effectively than an unregularized LSTM."*.

Greff et al. 2015 show that adding gaussian noise on the inputs in order to regularize the network hurts performance and training time.

To regularize the bLSTM Hammerla et al. used max-in norm (as advocated by Srivastava et al., who use a feed-forward network) and "**carry-over**" (a new technique that they introduced, cf. citation below). Though it doesn't explain the model performance as you can see on the figure in 1.2 Hyperparameters finetuning. They're the only authors cited in this report who use max-in norm, I don't think it's useful to use it if one already uses gradient clipping (Pascanu et al. 2012).
*"Training on **long sequences** has a further issue that is addressed in this work. If we use the approach outlined above to train a sufficiently large RNN it may "memorise" the entire input-output sequence implicitly, **leading to poor generalisation** performance. In order to avoid this memorisation we need to introduce "breaks" where the internal states of the RNN are reset to zero: after each mini-batch we decide to retain the internal state of the RNN with a **carry-over probability p carry**, and reset it to zero otherwise. This is a novel form of regularisation of RNNs, which should be useful for similar applications of RNNs."*. I thought forget gates were specifically built to do that (Gers et al. 1999)... This seems useful as, according to Hochreiter & Schmidhuber 1997, *"LSTM can learn to bridge minimal time lags in excess of **1000** discrete time steps by enforcing constant error flow through constant error carrousels"* and the average task of **PaHaW** is more than **2286** timesteps' long (i.e. 11,4s).

Although Gers et al. 1999 state *"**Weight decay does not work**. It could only slow down the growth of cell states indirectly by decreasing the overall activity in the network. We tested several weight decay algorithms (Hinton, 1986), (Weigend et al., 1991) without any encouraging results."*, Lipton et al. use **weight decay** to *"combat exploding gradients"*. Maybe this is because Lipton et al. use the forget gate introduced by Gers et al. (i.e. weight decay doesn't work alone but does with forget gate).
Lipton et al. use *target replication* in order to speed up the training and improve performance : they generate a target for each time step of their LSTM in order to backpropagate error from these intermediate targets. I don't understand how they generate these intermediate targets. The authors found that dropout decreases overfitting (as Chiu et al.), enabling them to double the size of each hidden layer. Their classification task is multi label diagnosis over clinical data.

We can't use teacher forcing because it requires a target for the hidden units.

1.4 Transfer Learning

Dai et al. 2015 use the weights of a seq2seq autoencoder's hidden layer to initialize the weights of another network which task is to classify the sequence. *"A significant property of* *the sequence autoencoder is that it is unsupervised, and thus can be trained with large quantities of unlabeled data to improve its quality. Our result is that additional unlabeled data can improve the generalization ability of recurrent networks. This is especially useful for tasks that have limited labeled data.".* This technique improves performance over the tasks sentiment analysis (IMDB and Rotten Tomatoes) and text classification (20 Newsgroups and DBpedia). Also this techniques improves the stability of the network : it's less sensitive to the hidden size. Some examples suggests that this technique handles long-time dependencies better then vanilla LSTM.

In the same way, Malhotra et al. 2017 train a Seq2Seq autoencoder over 30 different time-series datasets[1]. The difference with Dai et al. is that, instead of using the autoencoder's hidden layer to initialize another model, they use it as an embedding or feature extractor and run an SVM over it. The authors outperform state of the art on the majority of the datasets (22 over 30).
This last work is very interesting (but it requires a lot of time to experiment as it's very different from our model) and resembles those of the "CNN-feature-extractor" (Cf. Report #2).

1.5 Conclusion

Dropout seems to be the most widely used and effective tool to regularize the network and prevent it from overfitting. Although whether using it on recurrent or feedforward layers is still up to debate. We should bear in mind that works on recurrent dropout have been mostly tested on language modeling tasks which are quite different from ours. Applying feedforward dropout slightly improved our results (cf. 2.2 Model architecture & evaluation).
Gradient clipping seems to be the most widely used and effective tool to train the model and combat exploding gradients. Weight decay seems to be a controversial tool, as well as adding gaussian noise to the input.
The transfer learning with autoencoder's seems very promising but requires more time to implement a priori than trying other architectures like the one depicted in 1.1 Model Architectures et al..
A lot of different architectures have been proposed, and obviously, this isn't an exhaustive review. Intuitively, I think we should focus on deep/stacked architectures which are supposed to handle long sequences as ours, as well as architectures applied to similar classification tasks as our.
Training the model using only stochastic gradient descent and gradient clipping seems to work for now (cf. 2.2 Model architecture & evaluation).

[1] www.cs.ucr.edu/~eamonn/time_series_data/

2 Code

Cf. Git Commits.
We use the LSTM as defined in PyTorch[2], thus, for each element in the input sequence, each layer computes the following function :

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi})\\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf})\\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg})\\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho})\\
c_t &= f_t c_{(t-1)} + i_t g_t\\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

Where h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time t-1, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function. W_* are the weight matrices and b_* are the bias vectors.

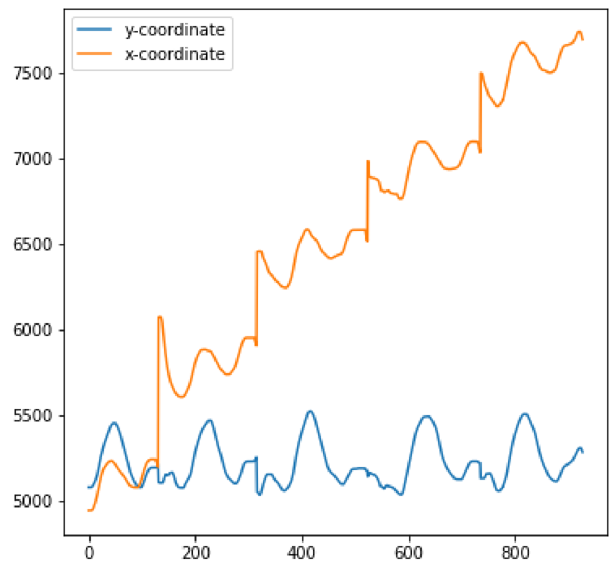
We currently feed the last hidden state of the LSTM (summed if bLSTM) to a FC layer with a single neuron which is used for binary classification after applying a Sigmoid to it.

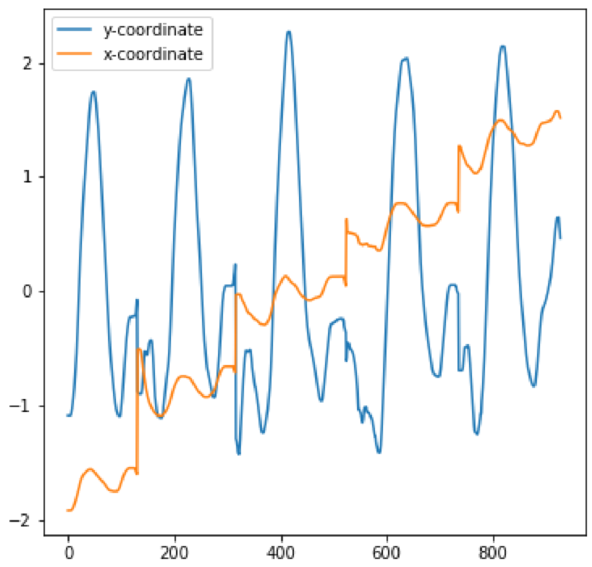
2.1 Data preprocessing

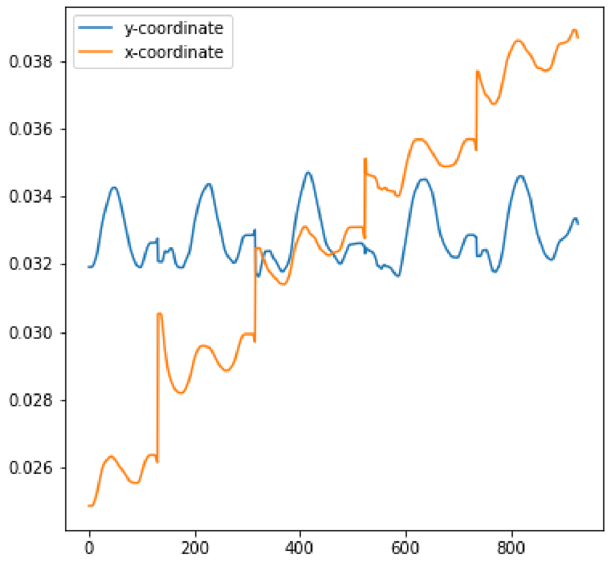
Standardizing a vector (e.g. x-coordinate of a task) gives it a mean of 0 and a standard deviation of 1. Normalizing a vector gives it a norm of 1.

Figure of the x and y coordinate recorded at 200Hz for the second task (writing multiple "l") of the first subject (PD) : raw, standardized and normalized (from left to right).

[2] https://pytorch.org/docs/stable/nn.html#torch.nn.LSTM







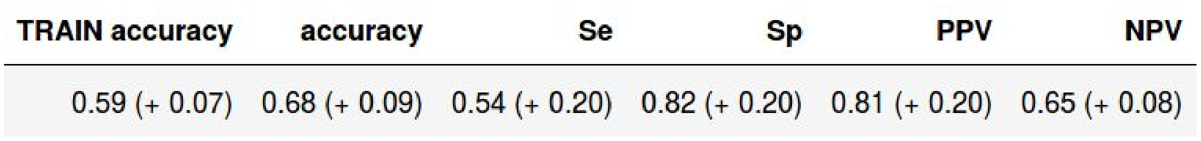
One can notice that the data ranges are preserved when normalizing unlike when standardizing. So I'd assume that it was better to normalize than to standardize but after experiment, the model won't fit the data if it's normalized, although it fits it quite well if it's standardized. After investigation I noticed that, between subjects, the normalization shifted the values of the data : you can see here that for the first subject the normalized x coordinate ranges between 0.026 and 0.038, whereas for the 26th subject, it ranges from 0.022 to 0.028 (although the raw ranges are similar). If we feed the raw data to the model (i.e. without any type of normalisation) it doesn't fit to the data : with *lr = 1e-2*, the loss converges at 0.7 (accuracy 0.469). Tried with *lr = 1e-3* (converges higher), *lr = 1e-1* (doesn't converge).

2.2 Model architecture & evaluation

With a very simple model :

- *learning_rate = 1e-3*
- *hidden_size=100*
- *num_layers=1*
- *bidirectional=False*
- *dropout=0.0*
- *no gradient clipping*
- → **43 701** total parameters, all *trainable*.

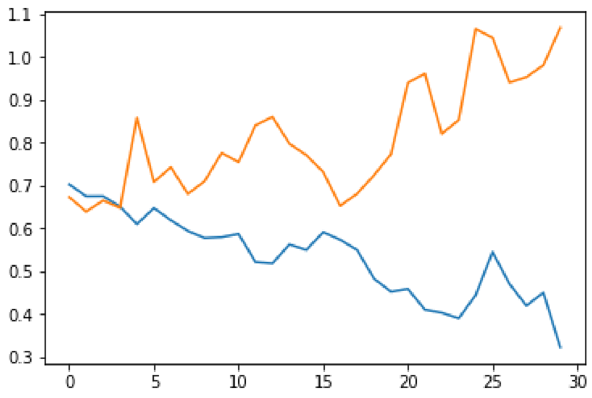
We're able to fit the training set (90% of the dataset) in 30 epochs until *loss = 0.323, accuracy = 0.906*
Although the model starts overfitting after 1, 2, 1, 4, 4, 2, 4, 3, 5 and 1 epochs (out of 10 folds, respectively).
Average metric over the 10 folds (+ standard deviation) after early stopping (i.e. we select the best epoch based on the validation accuracy for each fold) :

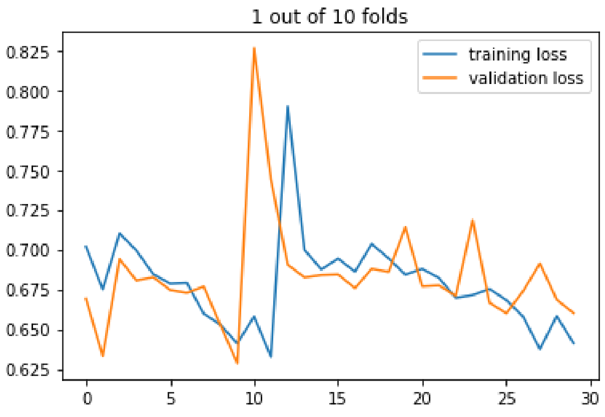


TRAIN accuracy	accuracy	Se	Sp	PPV	NPV
0.59 (+ 0.07)	0.68 (+ 0.09)	0.54 (+ 0.20)	0.82 (+ 0.20)	0.81 (+ 0.20)	0.65 (+ 0.08)

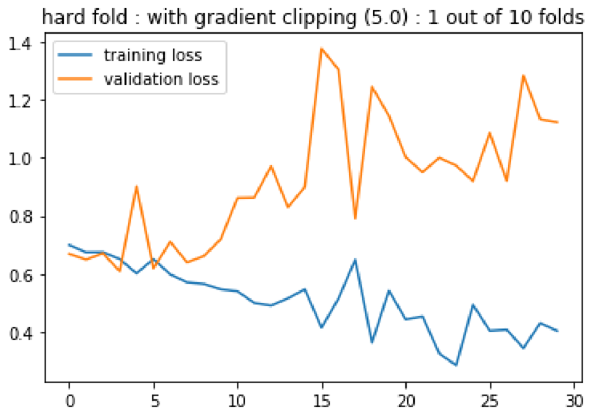
Every metric is for the validation set if not specified otherwise (i.e. except for the first column). Since the validation set is very small (~ 7 subjects) the metric standard deviation (std) is very high, this is consistent with Moetesum et al. (cf. Report #2). The accuracy for the spiral task is 76%, 63% and 55% for Moetesum et al., Drotar et al. and Impedovo et al., respectively. Notice that, in the same way as Drotar et al., we validate on the test set (i.e. on the validation fold). Therefore our results are overoptimistic and possibly overfitted to the dataset. It's unclear how Moetesum et al. trained their model (n° of epochs etc.).

If we use 5-fold CV with the same hyperparameters the model still fits the data in 30 epochs but less than with a 10-fold : *loss 0.438, accuracy 0.839.*
The validation metrics are quite poor in this case, thus the std is still very high. To allow for a fair comparison with Drotar et al. I think we should stick with 10 CV even though our model might be overfitted.
With these hyperparameters, on some folds the model won't fit the data very well (left : typical fold, right : hard fold) :





It may be because of exploding gradients : clipping the gradients norm to 5 helped with this, see below the loss' plot on the same fold as above, right.



Although it didn't help with the cross validation results, on the contrary, they are slightly inferior compared to the ones without gradient clipping. The differences between these results might also be explained by the random initialization of the weights of the network. I set the random seed after the first few experiments to avoid this bias.
We should also try other regularization techniques (cf. 1.3 Regularization & Overfitting**)**.

Following the guidelines of Greff et al. (cf. 1.2 Hyperparameters finetuning), we should be able to fine tune the hyperparameters independently, starting from :

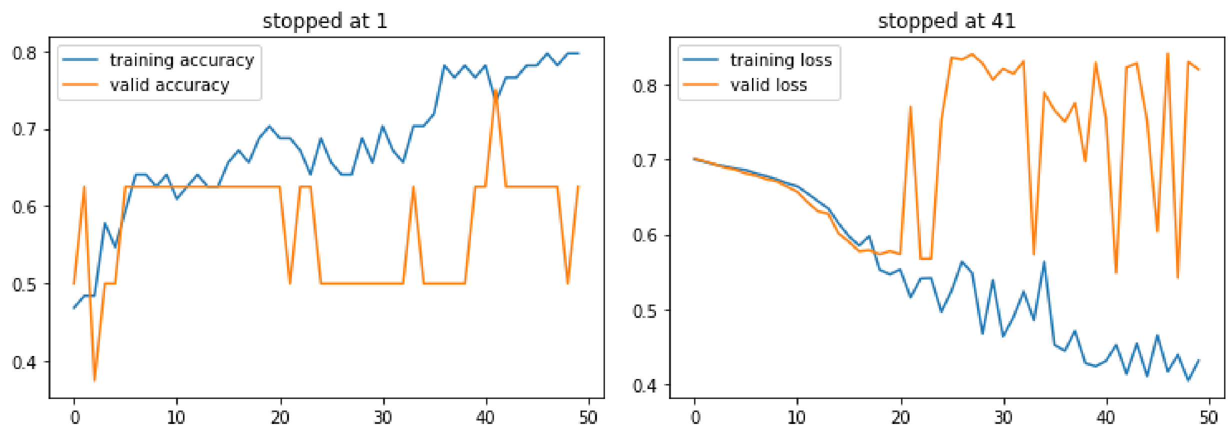
- learning rate
- hidden size
- number of layers
- regularization techniques (e.g. dropout, although Karpathy[3] and Lipton et al. suggests that it interacts with the hidden size so we might want to tune these hyperparameters together)

[3] https://github.com/karpathy/char-rnn

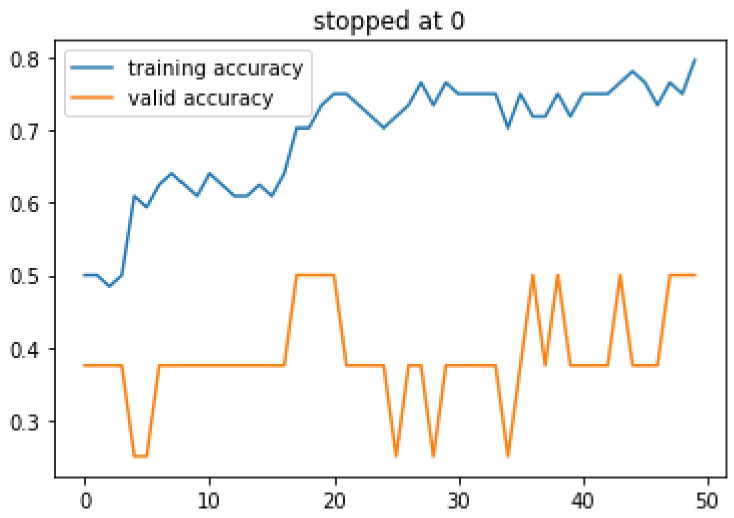
Throughout these experiments we shall use a bidirectional LSTM, which can only better the results in my opinion. Also the decoder architecture will be kept as simple as possible : one FC layer which is fed the last output of the LSTM (the sum of the 2 outputs from both directions). The FC layer outputs a single neuron which is fed to a Sigmoid layer in order to compute Binary Cross Entropy.
Once the above experiments are done, I will start to play with more original decoder architectures (cf. 1.1 Model Architectures et al.). I thought we might also use an SVM as Moetesum et al. or OPF as Passos et al. 2018 (cf. Report #2) but we can't because unlike them we need to train the network (they use a CNN pretrained on ImageNet). I should also try to train the model on different tasks.
All the results of the experiments are available in the git repo.
Note :

- the max epochs was set at 5 for the first 2 lines, 10 for the 3rd and 4rd lines and 50 for the rest
- there was no patience for early stopping for the lines 2 to 8 but no improvement was okay : you can see the differences between line 8 and 9. They are probably explained because when we tolerated the lack of improvement, the model could wait for tens of epochs (cf. "early stopped"), overfitting until "by chance" the accuracy would went up (cf. figure below : left for accuracy, right for loss). Thus the **results of lines 8 are overoptimistic**.
- the seed for random weights initialization was not fixed for the lines 2 to 8. Therefore a direct comparison with and between those might be biased.
- on the lines 2 to 12, all the biases were init randomly. You can see the difference on line 13 where we started to init the forget gate bias at 1 : it improved training accuracy, validation PPV and reduced the variance (std).

Model overfitting until (probably) random amelioration (line 8)



According to these first experiments, the best learning rate is 1e-3. With 1e-4 there is more variance (std) and on some folds the model won't fit the data (cf. figure below), we might try it again along with regularization techniques.
LR = 1e-4, model won't fit on some folds

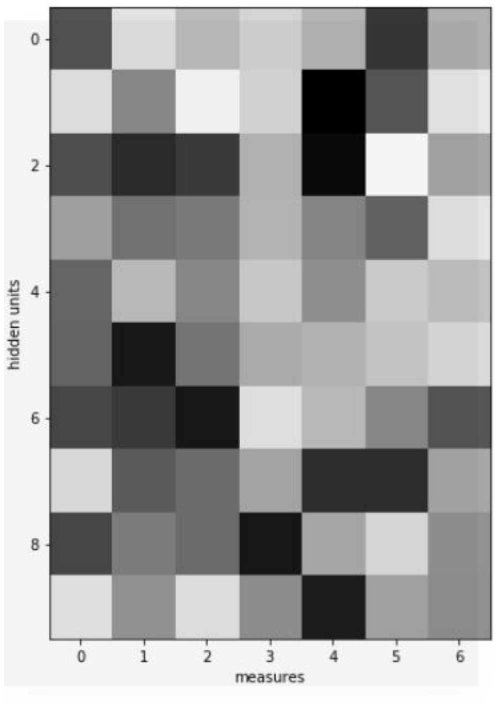


Downsampling the data 10 times doesn't help with the overfitting as you can see on line 14. Some extra analysis is available in the notebook's Visualization section.

2.3 Visualization & Interpretation

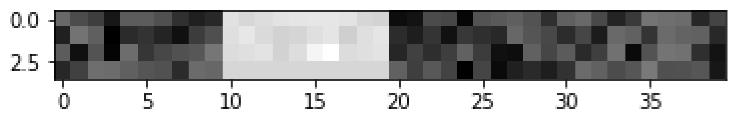
2.3.1 Input weights

Karpathy, Johnson et al. wrote a great article about visualizing RNN, however, their classification task (language modeling) allows for a simpler interpretation. I tried visualizing the input weights of a model fitted to the data in order to see which of the 7 measures was activated. As you can see below, the weights differ along the layer (i.e. between the LSTM units) without any measure standing out (I also tried plotting each measure along time or compute the mean of each measure). One possible interpretation is that there's no measure which is consistently "interesting" along the sequence. However I wasn't able to find any pattern when plotting the weights along time.
Plot of the weights of the 10 first input units (of the one and only layer of the network) :



2.3.2 Forget Gate

Interestingly, after using the trick of init the forget gate bias at 1, even after 22 epochs and an overfitted model, the bias of the forget gate are still worth ~1 (cf. figure below). One possible interpretation is that the model learns to *remember* the previous value of the *cell state*.



Conclusion - Todo List

The different LSTM architectures, regularization techniques are abundant, along with the hyperparameters in those. Moreover, there are many different ways we can represent the data / train the model (cf. Report #2's conclusion). Thus I'm not sure where to start… Intuitively, I thought about finishing the hyperparameters fine-tuning of our current model then to try out different data representation before trying out new architectures and hard-to-implement regularization…

References

Cooijmans, T., Ballas, N., Laurent, C., Gülçehre, Ç., & Courville, A. (2016). Recurrent batch normalization. arXiv preprint arXiv:1603.09025..pdf

Chiu, J. P., & Nichols, E. (2016). Named entity recognition with bidirectional LSTM-CNNs. Transactions of the Association for Computational Linguistics, 4, 357-370.

Dai, A. M., & Le, Q. V. (2015). Semi-supervised sequence learning. In Advances in neural information processing systems (pp. 3079-3087).

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). Learning to forget: Continual prediction with LSTM.

Graves, A. (2012). Supervised sequence labelling. In Supervised sequence labelling with recurrent neural networks (pp. 5-13). Springer, Berlin, Heidelberg.

Graves, A., Mohamed, A. R., & Hinton, G. (2013, May). Speech recognition with deep recurrent neural networks. In 2013 IEEE international conference on acoustics, speech and signal processing (pp. 6645-6649). IEEE.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2017). LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, *28*(10), 2222-2232.

Hammerla, N. Y., Halloran, S., & Plötz, T. (2016). Deep, convolutional, and recurrent models for human activity recognition using wearables. *arXiv preprint arXiv:1604.08880*.

Hermans, M., & Schrauwen, B. (2013). Training and analysing deep recurrent neural networks. In Advances in neural information processing systems (pp. 190-198).

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

Jozefowicz, R., Zaremba, W., & Sutskever, I. (2015, June). An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning* (pp. 2342-2350).

Karpathy, A., Johnson, J., & Fei-Fei, L. (2015). Visualizing and understanding recurrent networks. arXiv preprint arXiv:1506.02078.

Krueger, D., Maharaj, T., Kramár, J., Pezeshki, M., Ballas, N., Ke, N. R., ... & Pal, C. (2016). Zoneout: Regularizing rnns by randomly preserving hidden activations. arXiv preprint arXiv:1606.01305.

Lai, S., Xu, L., Liu, K., & Zhao, J. (2015, February). Recurrent convolutional neural networks for text classification. In Twenty-ninth AAAI conference on artificial intelligence.

Lipton, Z. C., Kale, D. C., Elkan, C., & Wetzel, R. (2015). Learning to diagnose with LSTM recurrent neural networks. arXiv preprint arXiv:1511.03677.

Malhotra, P., Ramakrishnan, A., Anand, G., Vig, L., Agarwal, P., & Shroff, G. (2016, June). LSTM-based encoder-decoder for multi-sensor anomaly detection. arXiv preprint arXiv:1607.00148.

Malhotra, P., TV, V., Vig, L., Agarwal, P., & Shroff, G. (2017). TimeNet: Pre-trained deep recurrent neural network for time series classification. arXiv preprint arXiv:1706.08838.

Ordóñez, F., & Roggen, D. (2016). Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. Sensors, 16(1), 115.

Pascanu, R., Mikolov, T., & Bengio, Y. (2012). Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, *2*.

Pascanu, R., Gulcehre, C., Cho, K., & Bengio, Y. (2013). How to construct deep recurrent neural networks. arXiv preprint arXiv:1312.6026.

Pham, V., Bluche, T., Kermorvant, C., & Louradour, J. (2014, September). Dropout improves recurrent neural networks for handwriting recognition. In 2014 14th international conference on frontiers in handwriting recognition (pp. 285-290). IEEE.

Salehinejad, H., Sankar, S., Barfett, J., Colak, E., & Valaee, S. (2017). Recent advances in recurrent neural networks. arXiv preprint arXiv:1801.01078.

Semeniuta, S., Severyn, A., & Barth, E. (2016). Recurrent dropout without memory loss. arXiv preprint arXiv:1603.05118.

Sønderby, S. K., & Winther, O. (2014). Protein secondary structure prediction with long short term memory networks. arXiv preprint arXiv:1412.7828.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1), 1929-1958.

Yao, K., Cohn, T., Vylomova, K., Duh, K., & Dyer, C. (2015). Depth-gated recurrent neural networks. *arXiv preprint arXiv:1508.03790*, *9*.